Three-Sum — Complete Notes & Step-by-Step Logic

Problem (exactly as provided):

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0. The solution set must not contain duplicate triplets.

Examples (exact as provided):

Example 1:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]

Example 2:
Input: nums = [0,1,1]
Output: []

Example 3:
Input: nums = [0,0,0]
Output: [[0,0,0]]

Constraints (exact as provided):

- 3 <= nums.length <= 3000
- -10^5 <= nums[i] <= 10^5

Original code (you provided):

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();

        vector<vector<int>> ans;

        sort(nums.begin(), nums.end());

        for(int i=0; i<n; i++) {

            if(i > 0 && nums[i] == nums[i-1]) continue;

            int j=i+1, k = n-1;

            while(j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                if(sum < 0) {
                    j++;

                } else if(sum > 0) {
                    k--;
                } else {
                    ans.push_back({nums[i], nums[j], nums[k]});
                    j++;
                    k--;

                    while(j < k && nums[j] == nums[j-1]) {
                        j++;
                    }
                }
            }
        }

        return ans;

    }
};
```

Full Step-by-Step Logic (detailed for revision):

1 **1. Sort the array (ascending).** This makes it possible to use the two-pointer technique and makes duplicate detection easy.
2 **2. Iterate i from 0 to n-3** (fix the first element of a potential triplet):
3 - If nums[i] > 0: break. Since array is sorted, all remaining numbers are >= nums[i], so sum cannot be 0.
4 - If i > 0 and nums[i] == nums[i-1]: continue. Skip duplicates to avoid repeating triplets that start with the same number.
5 **3. For each fixed i, use two pointers l and r:** set l = i + 1 and r = n - 1.
6 - While l < r: compute sum = nums[i] + nums[l] + nums[r].
7 • If sum < 0: move l++ (we need a larger sum because array is sorted).
8 • If sum > 0: move r-- (we need a smaller sum).
9 • If sum == 0: record the triplet {nums[i], nums[l], nums[r]}. Then move both pointers (l++ and r--) *and* skip duplicates on both sides:
10 - while (l < r && nums[l] == nums[l-1]) l++;
11 - while (l < r && nums[r] == nums[r+1]) r--;
12 **4. Continue until all i values are processed.** Return the collected triplets.

Pseudocode (concise):

```
sort(nums)
for i in range(0, n-2):
    if nums[i] > 0: break
    if i > 0 and nums[i] == nums[i-1]: continue
    l = i+1
    r = n-1
    while l < r:
        s = nums[i] + nums[l] + nums[r]
        if s < 0: l += 1
        elif s > 0: r -= 1
        else:
            add [nums[i], nums[l], nums[r]] to answer
            l += 1
            r -= 1
            skip duplicates at l and r
```

Corrected & commented C++ implementation:

```cpp
#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> ans;
        if (n < 3) return ans;

        sort(nums.begin(), nums.end()); // O(n log n)

        for (int i = 0; i < n - 2; ++i) {
            // Early stop: if the current number > 0, remaining can't sum to 0
            if (nums[i] > 0) break;

            // Skip duplicate 'i' values
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            int l = i + 1, r = n - 1;

            while (l < r) {
                int sum = nums[i] + nums[l] + nums[r];
                if (sum < 0) {
                    ++l; // need a larger sum
                } else if (sum > 0) {
                    --r; // need a smaller sum
                } else {
```

```
                    // found a valid triplet
                    ans.push_back({nums[i], nums[l], nums[r]});
                    ++l;
                    --r;
                    // skip duplicates for left and right pointers
                    while (l < r && nums[l] == nums[l - 1]) ++l;
                    while (l < r && nums[r] == nums[r + 1]) --r;
                }
            }
        }

        return ans;
    }
};
```

Complexity Analysis (step-wise justification):

- Sorting: O(n log n) time.
- Two-pointer search: For each i (O(n) iterations), the inner two-pointer loop runs in O(n) time total (each pointer moves at most n steps). So combined it's O(n^2).
- Total time: O(n log n + n^2) = O(n^2).
- Space: O(1) extra auxiliary space (only pointers and temporary variables), excluding the space required for the output triplets.

Example walkthrough (nums = [-1,0,1,2,-1,-4]):

- Sort: [-4, -1, -1, 0, 1, 2]
- i = 0 (nums[i] = -4): l = 1 (-1), r = 5 (2) → sum = -3 < 0 → l++ ... no triplet found for i=0.
- i = 1 (nums[i] = -1): l = 2 (-1), r = 5 (2) → sum = 0 → record [-1, -1, 2]. Move l -> 3 (0), r -> 4 (1).
- Now sum = -1 + 0 + 1 = 0 → record [-1, 0, 1]. Move l -> 4, r -> 3 (stop).
- i = 2 is duplicate of i = 1, so skip. Remaining i values don't produce new triplets. Final: [[-1,-1,2], [-1,0,1]].

Revision checklist & common pitfalls:

- Always sort the array first.
- Skip duplicate values for 'i' to avoid duplicate triplets.
- After finding a triplet, advance both l and r and skip duplicates on both sides.
- Use early break if nums[i] > 0 (micro-optimization).
- Be careful with arrays of many identical values (e.g., [0,0,0,...]).
- Don't forget to handle small n (e.g., n < 3) by returning empty result.

Memory aid (one-line):

Sort → Fix i → Two-pointer for -nums[i] → Skip duplicates → $O(n^2)$

Generated on: 2025-09-17 05:02:19 UTC